(12) DEMANDE INTERNATIONALE PUBLIÉE EN VERTU DU TRAITÉ DE COOPÉRATION EN MATIÈRE DE BREVETS (PCT)

(19) Organisation Mondiale de la Propriété Intellectuelle
Bureau international

(43) Date de la publication internationale
11 août 2005 (11.08.2005)

PCT

(10) Numéro de publication internationale
WO 2005/073859 A2

(51) Classification internationale des brevets[7] : G06F 11/28

(21) Numéro de la demande internationale :
PCT/FR2004/003273

(22) Date de dépôt international :
16 décembre 2004 (16.12.2004)

(25) Langue de dépôt : français

(26) Langue de publication : français

(30) Données relatives à la priorité :
0315633 31 décembre 2003 (31.12.2003) FR

(71) Déposant *(pour tous les États désignés sauf US)* : **TRUSTED LOGIC** [FR/FR]; 5, rue du Bailliage, F-78000 Versailles (FR).

(72) Inventeurs; et
(75) Inventeurs/Déposants *(pour US seulement)* : **BOLIGNANO, Dominique** [FR/FR]; 1, résidence les Tournelles, F-78810 Feucherolles (FR). **LEROY, Xavier** [FR/FR]; 37, Boulevard Saint Antoine, F-78000 Versailles (FR). **MARLET, Renaud** [FR/FR]; 35, rue Neuve, F-33000 Bordeaux (FR).

(74) Mandataire : **DE SAINT PALAIS, Arnaud**; Cabinet Moutard, 35, rue de la Paroisse, F-78000 Versailles (FR).

(81) États désignés *(sauf indication contraire, pour tout titre de protection nationale disponible)* : AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) États désignés *(sauf indication contraire, pour tout titre de protection régionale disponible)* : ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), eurasien (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), européen (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Déclaration en vertu de la règle 4.17 :
— *relative à la qualité d'inventeur (règle 4.17.iv)) pour US seulement*

Publiée :
— *sans rapport de recherche internationale, sera republiée dès réception de ce rapport*

En ce qui concerne les codes à deux lettres et autres abréviations, se référer aux "Notes explicatives relatives aux codes et abréviations" figurant au début de chaque numéro ordinaire de la Gazette du PCT.

(54) Title: METHOD FOR CONTROLLING PROGRAM EXECUTION INTEGRITY BY VERIFYING EXECUTION TRACE PRINTS

(54) Titre : PROCEDE DE CONTROLE D'INTEGRITE D'EXECUTION DE PROGRAMMES PAR VERIFICATION D'EMPREINTES DE TRACES D'EXECUTION

(57) Abstract: The inventive method for controlling a program execution integrity by verifying execution trace prints consists in updating the representative print of an execution path and/or data applied for a program execution, comparing the actual print value (dynamically calculated to an expected value (statistically fixed, equal to a value of the print if the program execution is not disturbed) at a determined program spots and in carrying out a particular processing by the program when the actual print differs from the expected value.

(57) Abrégé : Procédé de contrôle d'intégrité d'exécution de programmes par vérification d'empreintes de traces d'exécution mettant en oeuvre: - la mise à jour d'une empreinte représentative du chemin d'exécution et/ou des données manipulées lors de l'exécution du programme, - la comparaison de la valeur courante de l'empreinte (calculée dynamiquement) à une valeur attendue (fixée statiquement, égale la valeur que doit avoir l'empreinte si l'exécution du programme n'est pas perturbée) en des points déterminés du programme, - la réalisation d'un traitement particulier effectué par le programme au cas où l'empreinte courante diffère de la valeur attendue.

WO 2005/073859 A2

PROCEDE DE CONTROLE D'INTEGRITE D'EXECUTION DE PROGRAMMES PAR VERIFICATION D'EMPREINTES DE TRACES D'EXECUTION.

La présente invention a pour objet un procédé de contrôle d'intégrité d'exécution de programmes par vérification d'empreintes de traces d'exécution.

Elle s'applique notamment au contrôle d'intégrité d'exécution de programmes, ladite exécution de programmes mettant à jour une empreinte représentative du chemin d'exécution et/ou des données manipulées, sachant qu'en des points déterminés du programme, la valeur courante de l'empreinte est comparée à une valeur attendue.

Certains petits systèmes embarqués, comme par exemple les cartes à puce, sont conçus pour protéger les données et programmes qu'ils contiennent. En particulier, le support matériel de ces systèmes rend très difficile l'observation et la modification des données stockées, y compris lors de l'exécution des programmes.

La protection n'est toutefois pas totale. Ces systèmes peuvent être exposés à des actions malveillantes (aussi appelées « attaques ») qui visent à altérer le bon fonctionnement du système et des programmes, ou à dévoiler des informations confidentielles. Des attaques physiques (aussi dites matérielles) sont par exemple des perturbations de l'alimentation électrique, des bombardements par des rayonnements, la destruction de portes logiques, etc.

De telles attaques peuvent conduire à des dysfonctionnements du programme exécuté sur la carte : les instructions et les données lues depuis la mémoire peuvent être incorrectes, et le processeur peut exécuter incorrectement certaines instructions. De telles perturbations peuvent rendre invalide du code qui vérifie des conditions de sécurité avant d'opérer certaines actions sensibles.

Soit l'exemple (simplifié) du code suivant, écrit en langage C :

```
if (! condition1()) goto error;
if (! condition2()) goto error;
do_sensitive_action();
```

L'action sensible « do_sensitive_action() » n'est normalement exécutée que si les deux conditions « condition1() » et « condition2() » sont satisfaites. Néanmoins, des perturbations physiques peuvent amener le processeur à sauter directement par-dessus le code des deux « if » ou à lire depuis la mémoire une série d'instructions différentes du code des deux « if ». Dans ce cas, le processeur peut exécuter « do_sensitive_action() » même si les tests « condition1() » et « condition2() » ne sont pas satisfaits.

Une technique connue pour « durcir » un programme contre ce genre d'attaques consiste à introduire de la redondance entre le flot de contrôle du programme et les valeurs de certaines variables auxiliaires. En d'autres termes, on utilise des variables auxiliaires pour garder trace du fait que l'exécution est bien passée par toutes les vérifications de conditions de sécurité. Dans le cas de l'exemple ci-dessus, on pourrait écrire :

```
trace = 0;
if (condition1()) trace |= 1; else goto error;
if (condition2()) trace |= 2; else goto error;
if (trace == 3) do_sensitive_action();
```

Ainsi, si les deux premiers « if » ne sont pas exécutés correctement, il y a de bonnes chances que la variable « trace » ne soit pas égale à 3 à la fin, et que l'action « do_sensitive_action() » ne soit donc pas exécutée.

Même dans ce cas, une perturbation pourrait faire en sorte que le test « trace == 3 » soit toujours vrai ou soit lui aussi ignoré. De manière générale, pour la pertinence de l'invention présentée ici, on fait l'hypothèse que les perturbations causées par des attaques physiques sont relativement grossières : elles peuvent inactiver l'exécution d'une partie du programme ou changer le contenu apparent d'une partie de la mémoire, mais l'étendue de cette partie n'est pas contrôlable finement par l'attaquant.

Cette approche peut se généraliser de la manière suivante. Le programme tient à jour une valeur représentative des points de contrôle importants par lesquels passe son exécution. Cette valeur peut être une empreinte (un « hash »), par exemple une somme de contrôle (« checksum »), calculée sur des entiers qui caractérisent ces points de contrôle importants. Avant d'exécuter une opération sensible, cette valeur mise à jour en cours d'exécution peut être comparée à sa valeur normale (attendue), précalculée manuellement par le programmeur à partir de la structure de contrôle du programme. Par exemple :

```
trace = 42;
for (i = 0; i < 4; i++) {
        if (provided_pin[i] != actual_pin[i]) goto error;
        trace = h(trace, i);
}
if (trace == 2346) do_sensitive_action();
```

Il faut cependant noter que ce n'est pas ainsi qu'une vérification de code PIN est habituellement codée ; il s'agit juste ici d'un exemple illustratif de l'usage du calcul d'empreinte.

L'opérateur de hachage utilisé ici est la congruence linéaire « h(t,x) = ((33 * t) mod (2^16)) xor x » où :

- « a * b » représente le produit de « a » par « b »,
- « a mod b » représente le reste de la division de « a » par « b » (c'est-à-dire le modulo),
- « a ^ b » représente l'élévation de « a » à la puissance « b »,
- « a xor b » est l'opération « ou logique exclusif » sur les bits représentant « a » et « b ».

La valeur 2346 n'est autre que « h(h(h(h(42, 0), 1), 2), 3) », c'est-à-dire l'empreinte de l'exécution normale de la boucle (4 tours pour « i » variant de 0 à 3). Si la boucle n'est pas exécutée ou si elle se termine anormalement avant d'avoir fait 4 tours, la valeur de la variable « trace » avant l'appel de « do_sensitive_action() » sera très probablement différente de 2346.

On peut voir cet opérateur de hachage « h » comme un opérateur qui permet de calculer incrémentalement une fonction de hachage « H » qui opère sur des séquences d'entiers qui caractérisent lesdits points de contrôle importants. Cette fonction de hachage sur les séquences d'observations est définie par : « $H(T_0, i_1, i_2, \ldots, i_n) = h(\ldots h(h(T_0, i_1), i_2)\ldots, i_n)$ ».

Pour pouvoir être utilisée dans des petits systèmes embarqués disposants de faibles capacités de calcul, l'opérateur de hachage « h » doit être rapide à calculer. De plus, la fonction de hachage « H » résultante doit aussi être résistante aux pré-images : la probabilité pour que l'empreinte « $H(T_0, i_1, \ldots, i_n)$ » d'une séquence aléatoire « $T_0, i_1, \ldots, i_n$ » soit égale à une valeur donnée doit être faible (de l'ordre de 2^{-B} où B est le nombre de bits pour coder la valeur de l'empreinte.)

Dans certains cas décrits ci-dessous, il pourra aussi être utile de disposer d'un opérateur de hachage « h » qui soit inversible en son premier argument : pour toutes valeurs « t' » et « x », il existe « t » tel que « $t' = h(t, x)$ ».

On peut par exemple utiliser pour « h » une congruence linéaire, comme c'est le cas dans l'exemple donné ci-dessus. Cette congruence linéaire a de plus la propriété d'inversibilité. Plus généralement, on peut utiliser n'importe quelle fonction qui combine les opérations suivantes : addition, soustraction, ou logique exclusif (xor) avec une constante ou avec « x » ; rotation d'un nombre constant de bits ; multiplication par une constante impaire.

On peut utiliser également une fonction de contrôle de redondance cyclique (CRC), comme par exemple CRC-16 ; une telle fonction peut être calculée de manière efficace à l'aide de tables précalculées.

En revanche, même si une somme de contrôle (checksum, définie comme la somme des entiers passés en argument) a bien les bonnes propriétés de rapidité de calcul, de résistance aux pré-images, et d'inversibilité, elle est peu satisfaisante car la fonction de hachage obtenue est insensible à l'ordre des arguments.

À l'autre extrémité du spectre se trouvent des fonctions de hachage cryptographiques (digests, comme SHA-1, MD-5, etc.) qui sont très sûres mais aussi coûteuses à calculer.

La prise d'empreinte peut se décrire à l'aide de deux opérations principales d'affectation et de mise à jour d'empreinte : « setTrace(T) » fixe la valeur initiale de l'empreinte à « T » (typiquement, une valeur quelconque), et « addTrace(N) » ajoute à la trace un entier « N » caractéristique d'une observation de l'exécution du programme. Dans l'exemple ci-dessus, « setTrace(T) » correspond à l'affectation « trace = T; » et « addTrace(N) » correspond à la mise à jour de l'empreinte par « trace = h(trace, N) ».

La valeur « N » fournie lors d'une opération « addTrace(N) » peut elle-même être le résultat d'une fonction de hachage. Plus l'entier « N » est représentatif de l'exécution locale du programme, c'est-à-dire plus « N » varie en présence d'une perturbation possible de l'exécution, meilleur est le pouvoir de détection d'une attaque.

La vérification d'empreinte peut se décrire à l'aide d'une unique opération : « checkTrace(T) » vérifie que la valeur courante de l'empreinte est bien « T ». Dans le cas contraire, c'est une indication que le programme a subi une attaque. En toute rigueur, des systèmes comme des cartes à puce s'usent et une différence d'empreinte pourrait aussi être le signe d'une défaillance matérielle.

Dans tous les cas, le programme ne peut plus rendre le service pour lequel il est conçu. Dans des systèmes critiques, les mesures prises en cette circonstance consistent typiquement à sécuriser certaines données du programme et à interrompre, définitivement ou non, son exécution. Lorsque c'est matériellement possible, un signal sonore ou visuel peut en outre avertir l'utilisateur du dysfonctionnement.

Au vu de l'exemple ci-dessus, il est manifeste que la mise en œuvre de cette technique demande beaucoup d'efforts de la part du programmeur, d'une part pour insérer les étapes de calcul de l'empreinte, de l'autre pour calculer les valeurs attendues de l'empreinte aux points où elle devra être vérifiée.

Compte tenu des problèmes précédemment évoqués, l'invention a plus particulièrement pour but la généralisation de cette technique et son automatisation.

A cet effet, elle propose un procédé de contrôle d'intégrité d'exécution de programmes par vérification d'empreintes de traces d'exécution mettant en œuvre :

- la mise à jour d'une empreinte représentative du chemin d'exécution et/ou des données manipulées lors de l'exécution du programme,
- la comparaison de ladite empreinte (valeur courante, calculée dynamiquement) à une valeur attendue (fixée statiquement, égale la valeur que doit avoir l'empreinte si l'exécution du programme n'est pas perturbée) en des points déterminés du programme,
- la réalisation d'un traitement particulier dans le cas où l'empreinte courante diffère de la valeur attendue.

Ce procédé pourra prendre en compte :

- que ladite empreinte ne porte que sur des fragments de code critiques du programme,
- que les valeurs attendues de l'empreinte en des points donnés du programme sont déterminées par une analyse statique de programme qui modifie éventuellement le code de programme pour rendre les valeurs d'empreinte prévisibles.

Ce procédé et ses perfectionnements sont décrits plus en détail ci-après à titre d'exemples non limitatifs.

L'analyse de programme pour la détermination des valeurs d'empreinte attendues peut se décomposer en plusieurs étapes. On considère tout d'abord le cas d'une routine (méthode, sous-programme, etc.) du programme.

Dans le cas où toutes les instructions séparant le début de la routine d'un point de contrôle d'empreinte sont linéaires, c'est-à-dire ne contiennent pas de branchements, la valeur attendue de l'empreinte est simplement « $h(\ldots h(h(T_0, i_1), i_2)\ldots, i_n)$ » où « T_0 » est la valeur initiale de l'empreinte et où « $i_1, \ldots, i_n$ » sont les entiers représentatifs de l'exécution au divers points d'observation sur le chemin d'exécution.

Plus généralement, étant donné une décomposition d'une routine sous forme d'un graphe de blocs de base (basic blocks), si l'on connaît statiquement la valeur de l'empreinte au début d'un bloc de base, on connaît de la même manière sa valeur en tout point du bloc.

Lorsque plusieurs chemins d'exécution différents mènent en un même point de programme, il n'est pas possible de prédire « la » valeur d'empreinte attendue en ce point de programme puisqu'il y en a en fait plusieurs. Dans ce cas, on « égalise » les valeurs de l'empreinte aux points de jointure dans le flot de contrôle : sur chaque branche qui mène au point de jointure (sauf éventuellement une des branches), on ajoute à l'empreinte courante une valeur constante telle que l'empreinte résultante est la même pour chaque branche, une fois atteint le point de jointure. Cette addition peut se faire explicitement dans le code, par exemple par opération directe sur la valeur de l'empreinte ou par appel à une routine dédiée. Soit par exemple le programme suivant :

```
if (cond) {
        addTrace(1);
        ...
        addTrace(2);
} else {
        addTrace(3);
        ...
        addTrace(4);
}
```

Si l'empreinte a la valeur « T_0 » avant le « if », elle aura respectivement les valeurs « $h(h(T_0, 1), 2)$ » et « $h(h(T_0, 3), 4)$ » à la fin de chacune des deux branches de ce « if ». Pour établir un point de contrôle d'empreinte après ce « if », on peut égaliser les branches de la manière suivante :

```
if (cond) {
        addTrace(1);
        ...
        addTrace(2);
} else {
        addTrace(3);
        ...
        addTrace(4);
        adjustTrace(X);
}
checkTrace(Y);
```

avec les définitions suivantes :

- « adjustTrace(N) » ajoute l'entier « N » à la valeur courante de l'empreinte,
- « X » est égal à « $h(h(T_0, 1), 2) - h(h(T_0, 3), 4)$ »,
- « Y » est égal à « $h(h(T_0, 1), 2)$ ».

Ainsi, quel que soit le chemin d'exécution pris, l'empreinte vaudra toujours « Y » après le « if ». Le fait d'additionner la valeur « X » à la valeur courante de l'empreinte rend cette empreinte prévisible tout en préservant le fait qu'elle est difficile à falsifier. L'ajustement aurait pu être pratiqué de manière symétrique sur la première branche du « if » plutôt que sur la seconde.

Ce schéma s'applique aussi au cas des boucles dont le nombre d'itérations est indéterminé : à chaque tour de boucle, l'empreinte est ajustée pour revenir à la même valeur initiale.

L'opération d'ajustement d'empreinte « adjustTrace » s'ajoute aux opérations de mise à jour d'empreinte (« addTrace ») et d'affectation d'empreinte (« setTrace ») pour former l'ensemble des opérations de « prise d'empreinte ».

L'opération d'ajustement d'empreinte « adjustTrace(N) » n'a pas à être nécessairement une opération basée sur une opération d'addition du genre « trace = trace + N; ». Elle peut être plus généralement définie comme « trace = adjust(trace, N); » où « adjust(T, N) » est une fonction inversible en son premier argument : pour toute empreinte d'origine « T » et toute empreinte cible « T' », il suffit de savoir déterminer « N = delta(T, T') » tel que « adjust(T, N) = T' ».

On retrouve ainsi le cas de l'addition : « adjust(T, N) = T + N » et « delta(T, T') = T' – T ». Mais on peut aussi par exemple utiliser le « ou logique exclusif » : « adjust(T, N) = T xor N » et « delta(T, T') = T xor T' ».

L'ajustement d'empreinte « adjustTrace(delta(T, T')); » envoie l'empreinte « T » sur l'empreinte « T' » en préservant la difficulté de falsification.

Les gestionnaires d'exceptions (exception handlers) d'un programme rendent difficile la détermination statique des valeurs de l'empreinte. En effet, l'exécution peut se brancher au début du code du gestionnaire à partir de n'importe quelle instruction couverte par ce gestionnaire. La valeur de l'empreinte au début du code du gestionnaire n'est donc pas prévisible statiquement. La mise en place d'ajustements d'empreinte comme décrits ci-dessus pour toutes les instructions couvertes par un gestionnaire d'exception permet en principe d'assurer une valeur d'empreinte connue au début du gestionnaire. Cependant, comme le nombre d'ajustements nécessaires est grand, cela peut être coûteux en taille de code, en particulier si les ajustements

sont codés explicitement par des appels de routine du type « adjustTrace(N) ». En outre, les ajustements auraient comme effet secondaire de rendre l'empreinte constante sur tout l'intervalle du code couvert par un gestionnaire d'exception. L'empreinte serait donc alors inefficace pour détecter les perturbations d'exécution.

Une solution à ce problème est de forcer l'empreinte à une valeur connue statiquement lorsque l'exécution entre dans un gestionnaire d'exceptions. Cela peut être fait par l'interprète de la machine virtuelle, lorsqu'il traite le rattrapage d'une exception. Une autre possibilité est d'insérer un appel à une routine « setTrace(T) » au début de chaque gestionnaire d'exceptions, où « T » est une valeur choisie arbitrairement ; cette routine positionne la variable « trace » à la valeur de son argument.

Avec cette solution, la propriété de contrôle d'intégrité d'exécution est localement perdue entre le fragment d'exécution qui précède le branchement et celui qui le suit. Mais elle reste globalement préservée sur l'ensemble de l'exécution du programme.

Cette méthode peut plus généralement s'appliquer à tous les points de forte convergence du flot de contrôle de la routine.

Les subroutines, dans des langages comme ceux de la machine virtuelle « Java » (JVM) ou « Java Card » (JCVM) (marques déposées par Sun Mycrosystems), posent par exemple un problème similaire. Elles peuvent en effet être appelées depuis plusieurs points du code, avec des valeurs différentes pour l'empreinte. On peut les traiter de la même manière que les gestionnaires d'exception, en forçant la valeur de l'empreinte lorsqu'on entre dans une subroutine. Les points d'appels de subroutines étant bien délimités

(par exemple, instructions « jsr » de la JVM et de la JCVM) et généralement assez peu nombreux, on peut aussi les traiter comme des branchements normaux, avec insertion d'appels à « adjustTrace » avant les appels de subroutines pour garantir une même valeur de l'empreinte en tous les points d'appel d'une même subroutine.

Par ailleurs, le calcul d'empreinte et son contrôle peuvent toujours être limités à des fragments de code critiques du programme. Cela permet de réduire l'accroissement en taille du code pour introduire la redondance que représentent les opérations sur les empreintes. Cela réduit également le ralentissement dû aux calculs supplémentaires du programme.

Les opérations sur les empreintes ne modifient pas le comportement du programme (en l'absence de perturbation). Les opérations de mise à jour d'empreinte (du type « addTrace ») et d'ajustement d'empreinte (de type « adjustTrace ») se placent conceptuellement sur les arcs du graphe de flot de contrôle qui relie entre eux les points d'exécution du programme. De cette manière, la mise à jour d'empreinte et l'ajustement peuvent dépendre non seulement des points de programme parcourus en cours d'exécution mais aussi des branches spécifiques suivies, y compris dans le cas où plusieurs branches relient deux mêmes points. En pratique, dans le cas où les opérations de mise à jour et d'ajustement d'empreinte sont insérées effectivement dans le code exécutable du programme, elles sont disposées de manière à être exécutées en séquence entre les deux points de programme de l'arc sur lesquelles elles sont placées. Cette insertion peut nécessiter des modifications locales mineures du programme, comme l'ajout de branchements supplémentaires.

En revanche, les opérations d'affectation (de type « setTrace ») et de contrôle d'empreinte (de type « checkTrace ») se placent conceptuellement sur les

points de programme et non sur les arcs entre ces points. La valeur de l'empreinte est en effet une propriété des points de programmes : quels que soient les chemins d'exécution menant à un point de programme donné, la valeur courante de l'empreinte (calculée dynamiquement) doit être égale à une valeur fixe attendue (connue statiquement) ; et cette valeur attendue peut être forcée par une opération d'affectation d'empreinte.

La valeur de l'empreinte en chaque point d'exécution d'une routine de programme ainsi que les ajustements à insérer pour rendre prévisible la valeur de l'empreinte peuvent être déterminés automatiquement par une analyse statique de programme qui modifie éventuellement le code du programme pour rendre les empreintes prévisibles et pour les contrôler.

On se donne pour cela une routine du programme et des informations concernant la mise à jour d'empreinte (points de programme et nature des observations de l'exécution en ce point de programme), l'affectation d'empreinte (points de programme où l'empreinte doit être forcée à certaines valeurs) et éventuellement le contrôle d'empreinte (points de programme où l'empreinte doit être contrôlée).

Les informations de mise à jour, d'affectation et de contrôle d'empreinte peuvent par exemple être données par des insertions explicites de directives (commentaires, pragmas, code effectivement exécuté, etc.) dans le code du programme. De telles directives sont considérées de manière neutre du point de vue du flot de contrôle du programme : les directives de mise à jour d'empreinte de type « addTrace » se placent entre deux points d'exécution effectifs (hors directives) du programme, et ainsi sur les arcs qui relient conceptuellement entre eux les points d'exécution du programme ; les directives d'affectation (« setTrace ») et de contrôle d'empreinte

(« checkTrace ») portent sur le point d'exécution qui suit. Il en va de même pour les directives éventuellement insérées par la transformation de programme : les directives d'ajustement de type « adjustTrace » sont également neutres et se placent sur les arcs (pour corriger la valeur d'une empreinte avant d'atteindre un point de programme), c'est-à-dire entre deux points d'exécution.

Les directives d'affectation d'empreinte peuvent avoir été placées par une transformation de programme préalable. Cette transformation peut par exemple systématiquement placer une directive de type « setTrace » en tous les points de programme où conflue un nombre de branches d'exécution supérieur à un seuil donné. De telles directives peuvent également être placées systématiquement à tous les points d'entrée des subroutines et/ou des gestionnaires d'exception.

De même, les valeurs manipulées par les directives de mise à jour d'empreinte peuvent aussi avoir été déterminées par une transformation de programme préalable. En effet, les valeurs utilisées pour cette mise à jour (argument de « addTrace ») sont en général des constantes arbitraires. Une valeur quelconque (par exemple aléatoire) peut donc être automatiquement attribuée à chaque occurrence d'une telle directive. Dans certains cas, on peut aussi vouloir exploiter des invariants du programme. On procède alors à une analyse statique du programme afin de déterminer des invariants en certains points de programme, par exemple le fait que la somme de deux variables est constante le fait qu'une variable est inférieure à un certain seuil, le fait que le nombre de tours de boucle est connu, etc. Ensuite, au lieu de mettre à jour l'empreinte en fonction de constantes, on emploie des expressions qui, bien qu'elles portent sur des données dynamiques du programme, doivent avoir une valeur constante.

Le procédé de détermination automatique des valeurs d'empreinte attendues est ensuite défini par la séquence opératoire suivante.

- Initialisation de l'ensemble des points de programme à explorer au singleton constitué de la première instruction de la routine.
- Mémorisation au point d'entrée de la routine d'une valeur d'empreinte égale à la valeur initiale donnée de l'empreinte.
- Tant que ledit ensemble des points de programme à explorer n'est pas vide :
  - Extraction d'un point de programme (le point d'origine) dudit ensemble des points de programme à explorer.
  - Pour chacun des points de programme résultants possibles après exécution de l'instruction (les points cibles) :

    Si le point cible comporte une affectation d'empreinte et si ce point cible n'a pas encore été exploré, mémorisation au point cible de la valeur d'empreinte définie par l'affectation.

    Si le point cible ne comporte pas d'affectation d'empreinte et si ce point cible a déjà été exploré, insertion entre l'instruction au point d'origine et l'instruction au point cible d'un ajustement d'empreinte qui envoie la valeur de l'empreinte au point d'origine sur la valeur de l'empreinte mémorisée au point cible.

    Si le point cible ne comporte pas d'affectation d'empreinte et si ce point cible n'a pas encore été exploré, mémorisation au point cible de la valeur de l'empreinte au point d'origine, éventuellement modifiée par une mise à jour d'empreinte s'il y en a une entre le point d'origine et le point cible.

    Si le point cible n'a pas encore été exploré, ajout dudit point cible dans ledit ensemble des points de programme à explorer.

Certains langages ne permettent pas de manipuler facilement des directives neutres du type des directives de prise et de contrôle d'empreinte décrites ci-dessus. On peut employer dans ce cas une technique qui consiste à instrumenter le code du programme avec des instructions réelles.

Par exemple, on peut insérer dans le code du programme des appels de routine explicites, aussi bien pour la mise à jour d'empreinte et l'affectation d'empreinte que pour le contrôle d'empreinte. Par exemple, on peut écrire en « Java »:

```
{
        ...
        if (cond) {
                a[i+j] = b[i];
                addTrace();
        }
        checkTrace();
}
catch (Exception e) {
        setTrace();
        ...
}
```

Ce fragment de programme peut être transformé une première fois pour l'attribution de valeurs d'empreinte arbitraires pour les opérations de mise à jour et d'affectation :

```
{
        ...
        if (cond) {
                a[i+j] = b[i];
                addTrace(28935);
        }
        checkTrace();
}
catch (Exception e) {
        setTrace(9056);
        ...
}
```

Puis, le programme peut être transformé une seconde fois selon le procédé décrit ci-dessus pour déterminer les valeurs d'empreinte aux points de contrôle et pour insérer les ajustements nécessaires. Les valeurs attribuées par cette transformation dépendent de l'opérateur de hachage utilisé. Le fragment de programme donné en exemple ci-dessus pourra être transformé ainsi :

```
{
        ...
        if (cond) {
                a[i+j] = b[i];
                addTrace(28935);
                adjustTrace(16220);
        }
        checkTrace(13991);
}
catch (Exception e) {
        setTrace(9056);
        ...
}
```

Si l'on veut pouvoir compiler et exécuter le programme avant transformations, par exemple pour le mettre au point, il peut falloir adapter le programme source. En effet, si la librairie de prise et de contrôle d'empreinte ne comporte que des routines qui prennent en argument un entier, il est illégal d'écrire « checkTrace() ». Dans ce cas, on peut adopter la convention suivante : la valeur « 0 » (par exemple) dénote une valeur encore indéterminée. On écrit alors « checkTrace(0) » dans le code source initial. Après transformation, l'appel de routine deviendra par exemple « checkTrace(13991) ».

Cette technique est compatible avec la compilation du programme : les transformations de programme peuvent se faire aussi bien sur le code source que sur le code objet. Par exemple, en « Java Card », on peut écrire :

```
checkTrace(0) ;
```

Après compilation et conversion au format d'exécution de la JCVM, on a un code objet de la forme :

sconst_0

invokestatic checkTrace

Le procédé de détermination automatique de la valeur attendue de l'empreinte au moment du « invokestatic » peut s'appliquer au code objet de la JCVM et produire le code suivant :

sspush 13991

invokestatic checkTrace

Ce procédé de détermination automatique des valeurs d'empreinte peut être combiné à une analyse de programme qui effectue dans certaines cas un déroulage des boucles et récursions du programme. On peut alors calculer automatiquement des empreintes qui portent sur des variables du programme, comme dans l'exemple ci-dessus où « trace = h(trace, i) » figure à l'intérieur de la boucle d'indice « i ».

L'empreinte peut être calculée comme indiqué ci-dessus à partir de points d'observation donnés insérés explicitement dans le code du programme. L'empreinte peut aussi être calculée automatiquement et implicitement par l'interprète d'une machine virtuelle.

En particulier, l'empreinte peut porter sur le code des instructions exécutées par la machine virtuelle. Elle permet ainsi de contrôler que l'exécution successive des instructions du programme n'est pas perturbée.

La boucle principale de l'interprète d'une machine virtuelle a généralement la forme suivante :

```
while (1) {
        // Lecture du code de l'instruction à l'adresse pc
        opcode = *pc++;
        switch (opcode) {
        case INSTR :
                // Sémantique de l'instruction « instr »
        ...
        }
}
```

On peut instrumenter ce code ainsi :

```
while (1) {
        opcode = *pc++;
        addTrace(opcode);
        switch (opcode) {
        ...
        }
}
```

Pour le moment, on suppose que la variable « trace » associée à l'opération « addTrace » est, à chaque appel de routine, sauvegardée dans le bloc d'activation de la routine appelante et réinitialisée à une valeur connue « T_0 », et qu'elle est symétriquement restaurée depuis le bloc d'activation de l'appelant à chaque retour de routine. Ainsi, en un point quelconque du programme, la valeur de « trace » est « $T_n = h(...h(h(T_0, i_1), i_2)..., i_n)$ », qui est l'empreinte des codes opératoires « $i_1, ..., i_n$ » des instructions exécutées

depuis l'entrée dans la routine courante, en ignorant les instructions des méthodes appelées.

Cette mise à jour d'empreinte implicite peut être prise en compte explicitement et automatiquement par le procédé décrit ci-dessus de détermination automatique des valeurs d'empreinte attendues. Des invariants du programme visible au niveau de la machine virtuelle peuvent en outre être utilisés pour la mise à jour d'empreinte : hauteur de la pile d'opérande, présence de valeurs de certains types dans la pile d'opérande (si la pile est typée ou si les données permettent d'identifier leur type)

Le modèle de calcul d'empreinte implicite par une machine virtuelle se transpose directement au cas d'un microprocesseur exécutant du code natif. Le microprocesseur doit pour cela être conçu pour tenir à jour l'empreinte pour chaque instruction exécutée. Dans cette approche, la variable « trace » devient un registre spécial du processeur, accédé via des instructions de type « load » et « store » (pour sauver et restaurer la valeur de la trace à l'entrée et à la sortie des procédures), « addi » et « movi » (pour corriger et initialiser la valeur de la trace).

D'un point de vue pratique, la prise d'empreinte doit être réalisée de manière à ralentir au minimum le chemin critique du processeur. Ce peut être également un mode de fonctionnement débrayable du processeur, qui permet de n'effectuer les calculs d'empreinte que dans des sections critiques du code.

Un débrayage dynamique de ce type peut également être mis en œuvre dans le cadre d'un calcul d'empreinte implicite par une machine virtuelle.

Une autre possibilité pour accélérer le calcul d'empreinte est que le processeur dispose d'une instruction qui effectue l'opération de mise à jour « trace = h(trace, x); » en hardware. Cette instruction machine peut être

utilisée aussi bien dans la boucle d'interprétation d'une machine virtuelle que dans du code natif, où elle peut être insérée automatiquement par transformation du code machine. Le processeur peut comporter également des instructions dédiées pour effectuer les opérations d'ajustements, d'affectations et de contrôle d'empreinte.

De telles instructions spécialisées peuvent également être explicitement incluses dans le jeu d'instruction d'une machine virtuelle : au lieu d'opérer explicitement sur une variable « trace » ou de faire des appels de routine du type « addTrace(N) », « setTrace(T) », « adjustTrace(N) » et « checkTrace(T) », ces opérations peuvent être réalisées par des instructions dédiées de la machine virtuelle. Le calcul d'empreinte n'est pas dans ce cas implicite : il est explicite et nécessite une instrumentation (éventuellement automatique) du programme.

L'intérêt d'une telle approche est non seulement un gain en vitesse mais aussi en taille mémoire. Ainsi, par exemple, au lieu des six octets que requièrent l'appel « checkTrace(42) » dans du code objet JCVM, seuls trois octets (un pour l'instruction et deux pour l'argument entier court) seraient nécessaires.

Dans le cas où l'empreinte est mise à jour automatiquement et implicitement par la machine d'exécution (machine virtuelle ou processeur), plusieurs perfectionnements sont envisageables.

Tout d'abord, pour diminuer le surcoût en temps d'exécution dû à la prise d'empreinte, on peut envisager de ne pas modifier l'empreinte pour certaines classes d'instructions. Ceci peut s'effectuer à l'aide d'une table qui associe à tout code d'instruction un booléen qui indique si l'instruction est à tracer. La boucle principale de l'interprète s'écrit alors de la manière suivante :

```
while (1) {
        opcode = *pc++;
        if (update_trace[opcode])
                trace = h(trace, opcode);
        switch (opcode) {
        ...
        }
}
```

L'information des instructions à effectivement tracer (ici le tableau « update_trace ») peut accompagner le programme lors de son chargement et de son exécution sur la plate-forme d'exécution. Cette information peut en outre varier suivant les programmes et même suivant les différentes routines des différents programmes.

On peut également coder cette information « en dur » dans l'interprète pour les cas correspondants aux différents codes d'instruction. On a ainsi par exemple :

```
while (1) {
        opcode = *pc++;
        switch (opcode) {
        case INSTR1 : // tracée
                trace = h(trace, opcode);
                ...
        case INSTR2 : // pas tracée
                ...
        }
}
```

Pour détecter d'éventuelles perturbations sur la valeur des arguments immédiats des instructions, on peut également les intégrer dans le calcul de l'empreinte. Par exemple, dans le cas de la JCVM, pour l'instruction « bspush n » (qui pousse la constante entière « n » sur la pile d'opérande), l'interprète peut effectuer « trace = h(h(trace, BSPUSH), n) » au lieu de simplement « trace = h(trace, BSPUSH) ».

Pour conserver la possibilité de calculer statiquement les valeurs d'empreintes, ce schéma ne peut être appliqué qu'aux instructions dont les arguments immédiats ne sont pas modifiés au moment du chargement, à l'édition de lien. Dans le cas contraire, il faut être capable d'opérer sur le programme après édition de lien. C'est par exemple possible dans le cas où le programme est inclus en mémoire morte (ROM) dans un masque de carte à puce.

Pour rendre la valeur de l'empreinte encore plus sensible au flot de contrôle effectivement suivi à l'intérieur du code d'une routine, les instructions de

branchement conditionnel (par exemple « ifzero » dans la JCVM) et de branchement multiple (par exemple « stableswitch » dans la JCVM) peuvent mettre à jour la trace de manière différente suivant que le branchement conditionnel est pris ou non, ou suivant l'entrée de la table de saut qui est sélectionnée. Par exemple :

```
switch (opcode) {
case IFZERO:
    if (top_of_stack == 0) {
        trace = h(trace, BRANCH_TAKEN);
        pc += pc[0];
    } else {
        trace = h(trace, BRANCH_NOT_TAKEN);
        pc += 1;
    }
    ...
```

On peut aussi voir ce perfectionnement comme la prise en compte de la valeur connue de l'expression « top_of_stack == 0 » dans chacune des branches de l'instruction « ifzero », c'est-à-dire respectivement « true » et « false ».

Dans ce qui a été décrit jusqu'à présent, l'empreinte est calculée routine par routine. Il est inutile, et potentiellement coûteux en temps d'exécution, de mettre à jour la variable « trace » lors de l'interprétation de routines qui ne vont jamais consulter la valeur de cette variable, c'est-à-dire qui ne contiennent pas d'opérations de type « checkTrace ». L'information indiquant si une routine contient une opération « checkTrace » peut facilement être déterminé par un simple examen du code de la routine.

L'interprète de la machine virtuelle peut alors être modifié comme suit :

```
while (1) {
        opcode = *pc++;
        if (method_flags & NEEDS_TRACE)
                trace = h(trace, opcode);
        switch (opcode) {
        ...
        }
}
```

Dans le cas particulier de la JCVM, cette information peut par exemple être calculée un fois pour toute (par exemple au chargement du programme) et enregistrée sous forme d'un bit dans le champ « flags » de la structure « method_info ».

Jusqu'ici, on a considéré l'empreinte d'exécution comme locale à chaque méthode : la variable « trace » est préservée à l'appel de routine, et sa valeur représente une empreinte des points d'observation de l'exécution à l'intérieur de la routine courante, en excluant les routines appelées. La variable « trace » peut être rendue globale, et donc contenir une empreinte de toutes les instructions exécutées, y compris à travers des appels de routines, depuis des points d'entrée connus (par exemple, les méthodes « process » et « install » d'une applet « Java Card »), où « trace » a été initialisée à des valeurs connues.

Les motivations sont doubles : d'une part la possibilité de vérifier en une seule opération « checkTrace » l'absence de perturbations dans l'exécution

complète du programme, et non pas juste dans l'exécution de la routine où se trouve le « checkTrace » ; et d'autre part, dans le cas où l'empreinte est prise implicitement par la machine d'exécution, la simplification de cette machine (plus besoin de sauvegarder et de restaurer « trace » à chaque appel).

La principale difficulté de cette approche est le calcul statique des valeurs de l'empreinte en tout point du programme. Ce calcul reste possible, mais exige ou bien une analyse du programme complet (interprocédurale), ou bien l'établissement d'invariants sur les valeurs de l'empreinte au début et à la fin de chaque routine.

Dans l'approche par analyse globale, on suppose connaître, au moment de l'analyse statique, le code de toutes les routines pouvant être appelées directement ou indirectement à partir des points d'entrée du programme, y compris les routines appartenant à d'autres programmes, en particulier celles de librairies.

Sous cette hypothèse, le procédé de détermination statique de l'empreinte décrit ci-dessus s'étend comme suit : au lieu d'analyser et de transformer chaque routine individuellement, on les transforme toutes en même temps, en traitant les instructions d'appel de routines (« invoke », « call », etc.) comme des branchements inconditionnels à la première instruction de la routine appelée, et les instructions de retour d'appel (« return », etc.) comme des branchements vers les instructions suivant immédiatement l'appel correspondant.

En d'autres termes, on travaille non plus sur le graphe de flot de contrôle de la routine, mais sur le graphe de flot de contrôle interprocédural, contenant des arcs supplémentaires correspondant aux instructions d'appel et de retour de routine. On notera que pour les instructions qui déterminent dynamiquement la

routine à appeler effectivement (comme par exemple l'appel de méthode virtuelle ou de méthode d'interface), on ajoute des arcs de et vers toutes les routines en lesquelles l'appel peut se résoudre dynamiquement. Par exemple, dans la JVM, une approximation simple de l'ensemble des méthodes cibles de « invokevirtual C.m », est l'ensemble des méthodes « m » définies dans la classe « C » ou l'une de ses sous-classes. (On peut obtenir de meilleures approximations de cet ensemble à l'aide d'analyses statiques de flot de contrôle.)

Le procédé décrit ci-dessus dans le cas d'un routine, appliqué à ce graphe interprocédural, va insérer les ajustements d'empreintes nécessaires pour que la valeur de l'empreinte soit la même à tous les sites d'appel d'une même routine, et à la fin de toutes les routines susceptibles d'être appelées depuis le même site d'appel.

Le problème de l'approche globale ci-dessus est que l'hypothèse selon laquelle on connaît le code de toutes les routines est parfois trop forte : on connaît le code des méthodes du package en cours de transformation, mais pas forcément celui des routines de librairies ou partagé avec d'autres programmes.

Une première solution est de préserver la valeur de l'empreinte à travers les appels à des routines qui ne font pas partie du programme, par exemple en sauvant et restaurant la variable « trace » autour de ces appels.

Une autre solution, qui permet également de faire l'économie de l'analyse de flot interprocédurale, consiste à établir le « contrat » suivant entre toutes les routines : pour chaque routine de nom « r », si la valeur de l'empreinte est « E(r) » à l'entrée dans la routine, alors la valeur de l'empreinte est « S(r) » à

la sortie de la routine. Ici, « E » et « S » sont des fonctions associant des valeurs d'empreinte arbitraires à des noms de routines, par exemple calculées par hachage cryptographique. La signature de la routine (type de ses arguments et de la valeur de retour) peut aussi être associée au nom, notamment si elle permet de distinguer des routines sans rapport entre elles.

Le « contrat » ci-dessus est simple à assurer : avant chaque instruction d'appel à une routine « r », on insère un ajustement de l'empreinte pour l'amener à la valeur « E(r) » ; et pour chaque instruction de retour d'appel, on insère de même un ajustement de l'empreinte pour l'amener à la valeur « S(r) ».

Ce principe s'étend au cas de méthodes déterminées dynamiquement au moment de l'appel. Par exemple, dans le cas de « Java » ou de « Java Card », on peut employer des paires « (m, s) » où « m » est un nom de méthode et « s » une signature de méthode. Des hachages de cette paire permettent de définir « E(m,s) » et « S(m,s) ». On peut noter que si une méthode virtuelle en redéfinit une autre, ces deux méthodes ont même nom et même signature, donc mêmes valeurs « E » et « S ». Le « contrat » sur les empreintes à l'appel et au retour de méthodes s'applique ainsi de même manière.

Dans la mesure où seuls sont pris en compte les noms des routines (et éventuellement leur signature), la détermination des valeurs d'empreinte peut s'effectuer routine par routine, sans connaître l'intégralité du programme. La sortie prématurée d'une routine via une exception non rattrapée ne pose pas de problème supplémentaire, sous l'hypothèse que la machine virtuelle (implicitement) ou le gestionnaire d'exception (explicitement) réinitialise toujours l'empreinte à une valeur constante connue lorsqu'une exception est déclenchée.

Au lieu de demander au programmeur d'indiquer explicitement dans le source les endroits où une vérification de l'empreinte doit être effectuée (par exemple avec des appels explicites à « checkTrace() »), ces vérifications peuvent aussi être insérées automatiquement par un outil de transformation du code.

Par exemple, dans le cas de la JCVM, on peut effectuer un contrôle :

- avant chaque instruction qui écrit en mémoire persistante (EEPROM) : écriture dans un champ de classe (« putstatic ») ou de d'instance (« putfield »), écriture dans un tableau (« x-astore »),
- au début et à la fin de chaque transaction,
- avant l'appel de certaines méthodes de librairie, etc.

Cela permet notamment de mettre en place une politique de contrôle d'intégrité d'exécution qui considère que le programme peut exécuter n'importe quelles opérations, éventuellement perturbées, tant qu'il ne cherche pas à modifier la valeur des données persistantes et à faire des entrées-sorties.

La stratégie d'insertion automatique des vérifications d'empreintes est facile à changer, puisqu'elle affecte uniquement la phase de transformation du code. Elle résulte en pratique d'un compromis à trouver entre la fréquence des vérifications et l'augmentation de la taille du code et du temps d'exécution dus aux opérations « checkTrace ».

Plutôt que de modifier explicitement le programme par insertion d'instructions dans le code (par exemple pour appeler des routines du type « addTrace », « checkTrace », « adjustTrace » et « setTrace »), les données de prise et de contrôle d'empreinte peuvent être stockées dans des tables indépendantes au code exécutable du programme. Ces données peuvent comprendre les points

de programme où une opération d'empreinte est effectuée ainsi que les valeurs associées (les arguments de ces opérations). C'est la machine d'exécution (par exemple une machine virtuelle) qui se charge alors d'effectuer ces opérations lorsque le point de programme courant passe par les points de programme mémorisés dans les tables.

Plutôt que l'argument effectif d'une opération d'empreinte, les tables peuvent aussi stocker des valeurs qui indiquent quelles données dynamiques prendre en compte dans l'opération. Par exemple, une entrée complexe correspondant à « addTrace » peut indiquer de mettre à jour l'empreinte avec la valeur courante d'une variable ou d'un registre de la machine (parce que sa valeur est connu statiquement).

Un avantage de cette approche est de réduire l'encombrement mémoire des informations de calculs et de contrôle d'empreinte. En revanche, elle peut être coûteuse en temps d'exécution du fait des parcours de table. En cas de problème de performance, on peut employer une approche mixte : certaines opérations sont calculées automatiquement par la machine d'exécution ; d'autres opérations sont insérées explicitement dans le code du programme ; d'autres opérations encore sont stockées dans des tables. Par exemple, l'opération « addTrace » peut être calculée par une machine virtuelle ; les opérations « checkTrace » et « setTrace » peuvent être exprimées par des appels de routine explicites ; et « adjustTrace » peut être implémentée par une recherche en table. De plus, pour des raisons d'efficacité, la recherche en table pour « adjustTrace » peut n'être effectuée que dans le cas où un branchement est pris (en supposant les ajustements effectifs pratiqués uniquement dans ces cas-là).

Dans la mesure où les opérations de calculs et de contrôle d'empreinte ne sont pas standardisées, un programme instrumenté comme décrit ci-dessus pour le contrôle d'intégrité d'exécution peut ne pas fonctionner sur les plates-formes d'exécution qui ne sont pas adaptées.

Dans le cas où le code fait explicitement apparaître des appels de routine sur les empreintes (du type « addTrace », « checkTrace », « adjustTrace » et « setTrace »), le programme peut être accompagné d'une librairie qui implémente ces routines. Il sera alors portable sur toutes les plates-formes d'exécution. Toutefois, si la plate-forme dispose d'une implémentation plus efficace, elle peut la substituer à celle fournie avec le programme

Dans le cas d'un codage des prises et contrôles d'empreintes dans des tables, le programme est davantage portable. Par exemple, dans le cas de d'une JCVM, les tables peuvent être fournies sous forme de composants personnalisés supplémentaires : soit la machine les reconnaît et les exploite ; soit elle ne les reconnaît pas et les ignore (et dans ce cas ne pratique pas de vérification d'intégrité d'exécution).

Revendications

1. Procédé de contrôle d'intégrité d'exécution d'un programme par vérification d'empreintes de traces d'exécution, caractérisé en ce qu'il comprend :
   - la mise à jour d'une empreinte représentative du chemin d'exécution et/ou des données manipulées lors de l'exécution du programme,
   - la comparaison de ladite empreinte (valeur courante, calculée dynamiquement) à une valeur attendue (fixée statiquement, égale la valeur que doit avoir l'empreinte si l'exécution du programme n'est pas perturbée) en des points déterminés du programme,
   - la réalisation d'un traitement particulier dans le cas où l'empreinte courante diffère de la valeur attendue.

2. Procédé selon la revendication 1, caractérisé en ce que le traitement particulier du programme dans le cas où l'empreinte courante diffère de la valeur attendue consiste à sécuriser certaines données et/ou à avertir l'utilisateur du dysfonctionnement par un signal sonore ou visuel et/ou à interrompre, définitivement ou non, l'exécution dudit programme.

3. Procédé selon la revendication 1, caractérisé en ce que ladite empreinte ne porte que sur des fragments de code critiques du programme et/ou dans des états du programme jugés critiques.

4. Procédé selon la revendication 1, caractérisé en ce que ladite empreinte est calculée incrémentalement le long du chemin d'exécution du programme par composition successive d'une fonction dont un argument est la valeur courante de l'empreinte et un autre argument est une donnée d'observation spécifique au point et au moment de mise à jour de l'empreinte (état du

programme et/ou du point d'exécution du programme et/ou des données manipulées).

5. Procédé selon la revendication 4, caractérisé en ce que ladite fonction consiste en l'une des fonctions suivantes : somme de contrôle (« checksum »), congruence linéaire, contrôle de redondance cyclique (CRC), hachage cryptographique (« digest »), ou combinaison des opérations suivantes : addition, soustraction, « ou » logique exclusif (« xor ») avec une constante ou avec ladite donnée d'observation ; rotation d'un nombre constant de bits ; multiplication par une constante impaire.

6. Procédé selon la revendication 1, caractérisé en ce que l'empreinte est ajustée le long des chemins d'exécution avant d'atteindre certains points de convergence du flot de contrôle de manière à rendre égales les empreintes des chemins convergents.

7. Procédé selon la revendication 6, caractérisé en ce que l'opération d'ajustement consiste en une combinaison des fonctions suivantes : affectation à une valeur constante, addition avec une constante, « ou » logique exclusif (« xor ») avec une valeur constante.

8. Procédé selon la revendication 1, caractérisé en ce que, en certains points du programme, l'empreinte est affectée à une certaine valeur plutôt que déduite de la valeur d'empreinte précédente.

9. Procédé selon la revendication 8, caractérisé en ce que lesdits points de programme sont ceux où conflue un nombre de branches d'exécution supérieur à un certain seuil et/ou ceux qui sont les points d'entrée de

subroutines et/ou de gestionnaires d'exception, et en ce que ladite valeur affectée est une valeur donnée et/ou une valeur quelconque déterminée par tirage aléatoire et/ou une expression de programme déterminée par une analyse préalable comme invariante au point de programme considéré.

10. Procédé selon la revendication 1, caractérisé en ce que la valeur de l'empreinte est comparée à la valeur attendue en des points du programme déterminés par leur caractéristique particulière dans le graphe de flot de contrôle dudit programme et/ou par la nature des opérations effectuées auxdits points de programme.

11. Procédé selon la revendication 10, caractérisé en ce que lesdits points de programme sont situés après chaque branchement et/ou avant chaque point de jointure du flot de contrôle et/ou avant chaque opération qui écrit en mémoire persistante et/ou avant certaines opérations cryptographiques et/ou avant l'appel de certaines routines de librairie et/ou après l'appel de certaines routines de librairie.

12. Procédé selon l'une des revendications 1 à 11, caractérisé en ce que la prise d'empreinte (calcul et/ou mise à jour et/ou ajustement et/ou affectation) et/ou le contrôle d'empreinte sont réalisés :

- explicitement par une instrumentation du code du programme, et/ou
- explicitement par la machine d'exécution (machine virtuelle et/ou processeur de la plate-forme d'exécution), sur la base d'informations complémentaires au programme qui indiquent à ladite machine d'exécution en quels points de programme et/ou avec quelles valeurs (y compris des valeurs résultant d'opérations complexes) effectuer des opérations de prise et/ou de contrôle d'empreinte, et/ou

- implicitement par la machine d'exécution (machine virtuelle et/ou processeur de la plate-forme d'exécution), sur la base d'une observation particulière des instructions exécutées.

13. Procédé selon la revendication 12, caractérisé en ce que ladite instrumentation du code du programme est basée sur la manipulation explicite d'une variable ou d'un registre représentant l'empreinte et/ou sur l'appel à des routines spécialisées et/ou sur l'emploi d'instructions spécialisées de la machine d'exécution.

14. Procédé selon la revendication 12, caractérisé en ce que lesdites informations complémentaires sont codées dans des tables qui associent des points de programmes à un code définissant l'opération à réaliser et qui ne sont consultées par la machine d'exécution que lors de l'exécution d'instructions particulières.

15. Procédé selon la revendication 14, caractérisé en ce que lesdites instructions particulières sont des branchements et/ou des écritures en mémoire persistante et/ou des appels à certaines routines et/ou certaines opérations cryptographiques.

16. Procédé selon la revendication 1, caractérisé en ce que les valeurs attendues de l'empreinte et des ajustements d'empreinte en des points donnés du programme sont déterminées par une analyse statique du code du programme (source ou objet) qui peut simuler le déroulage de certaines boucles et récursions et qui peut modifier le code du programme pour rendre les valeurs d'empreinte prévisibles et/ou pour les contrôler.

17. Procédé selon l'une des revendications 4 à 16, caractérisé en ce que sont fournies à ladite analyse des informations concernant la mise à jour d'empreinte (points de programme et nature des observations de l'exécution en ce point de programme) et/ou l'ajustement d'empreinte (points de programme où l'empreinte doit être ajustée à une certaine valeur) et/ou l'affectation d'empreinte (points de programme où l'empreinte doit être forcée à une valeur) et/ou le contrôle d'empreinte (points de programme où l'empreinte doit être contrôlée), ces informations :

- étant déterminées automatiquement conformément au procédé selon l'une des revendications 6 à 11, et/ou
- étant données sous forme de directives consistant en des instructions placées dans le code du programme et opérant sur l'empreinte (telles que des appels de routine, prenant ou non en argument un entier quelconque) et/ou étant fournies sous forme de tables complémentaires au programme,
- et pouvant être complétées et/ou modifiées selon les valeurs calculées par ladite analyse.

18. Procédé selon la revendication 17, caractérisé en ce que, pour chaque routine du programme, les valeurs d'empreinte attendues sont déterminées par la séquence opératoire suivante :

- Initialisation de l'ensemble des points de programme à explorer au singleton constitué de la première instruction de la routine.
- Mémorisation au point d'entrée de la routine d'une valeur d'empreinte égale à la valeur initiale donnée de l'empreinte.
- Tant que ledit ensemble des points de programme à explorer n'est pas vide :

- Extraction d'un point de programme (le point d'origine) dudit ensemble des points de programme à explorer.
- Pour chacun des points de programme résultants possibles après exécution de l'instruction (les points cibles) :
  * Si le point cible comporte une affectation d'empreinte et si ce point cible n'a pas encore été exploré, mémorisation au point cible de la valeur d'empreinte définie par l'affectation.
  * Si le point cible ne comporte pas d'affectation d'empreinte et si ce point cible a déjà été exploré, insertion entre l'instruction au point d'origine et l'instruction au point cible d'un ajustement d'empreinte qui envoie la valeur de l'empreinte au point d'origine sur la valeur de l'empreinte mémorisée au point cible.
  * Si le point cible ne comporte pas d'affectation d'empreinte et si ce point cible n'a pas encore été exploré, mémorisation au point cible de la valeur de l'empreinte au point d'origine, modifiée par une mise à jour d'empreinte s'il y en a une entre le point d'origine et le point cible.
  * Si le point cible n'a pas encore été exploré, ajout dudit point cible dans ledit ensemble des points de programme à explorer.

19. Procédé selon l'une des revendications 12 à 18, caractérisé en ce que d'une part l'empreinte porte sur l'exécution complète du programme (y compris avec appels de routines) à partir de ses points d'entrée et d'autre part le procédé selon la revendication 17 est appliqué à un ensemble de routines en traitant les instructions d'appel statique de routine comme des branchements inconditionnels à la première instruction de la routine appelée, les instructions d'appel dynamique de routine comme des branchements conditionnels à la première instruction de la routine appelée correspondante, et les instructions de retour d'appel comme des

branchements vers les instructions suivant immédiatement l'appel correspondant.

20. Procédé selon la revendication 12, caractérisé en ce que le programme et/ou la machine d'exécution sont instrumentés pour que l'empreinte soit sauvegardée à l'appel de certaines routines (telles que celles qui ne font pas partie du programme ou qui ne sont pas analysables) et restaurée au retour de l'appel.

21. Procédé selon la revendication 12, caractérisé en ce que le programme et/ou la machine d'exécution sont instrumentés pour que l'empreinte soit ajustée à l'appel et au retour de certaines routines (y compris dans le cas de routines déterminées dynamiquement au moment de l'appel) de manière à être égale :

- à l'entrée de la routine appelée : à une valeur qui dépend du nom et/ou de la signature de la routine appelée (telle qu'une valeur obtenue par hachage cryptographique du nom et/ou de la signature) ;
- après retour dans la routine appelante : à une valeur qui dépend de la même manière du nom et/ou de la signature de la routine appelée, chaque gestionnaire d'exception concerné par l'appel de routine (c'est-à-dire pouvant être atteint lors de la levée d'une exception dans la routine appelée) devant affecter l'empreinte à une valeur déterminée.

22. Procédé selon l'une des revendications 3 et 12, caractérisé en ce que, dans le cas où l'empreinte est mise à jour implicitement par une machine d'exécution :

- la prise d'empreinte peut être temporairement suspendue pour éviter des calculs inutiles lors de l'exécution de fragments de code non

critiques du programme et/ou dans des états du programme jugés non critiques et/ou lors de l'exécution de certaines routines n'effectuant pas de contrôle d'empreinte ;

- la prise d'empreinte, lorsqu'elle n'est pas suspendue, porte sur chaque instruction exécutée,
  * y compris certains de ses arguments immédiats et/ou certains des invariants du programme pour cette instruction (tels que la hauteur de la pile d'opérande ou la présence de valeurs de certains types dans la pile d'opérande) et/ou les choix de branches effectués dans le cas où l'instruction est un branchement,
  * mais à la condition que l'instruction exécutée appartienne à une classe donnée des instructions à observer, ladite classe étant fixe pour la machine d'exécution ou bien donnée par une table associant à tout code d'instruction un booléen indiquant si l'instruction est à observer, et ladite table étant spécifique à différentes routines et/ou différents programmes.

23. Procédé selon la revendication 12, caractérisé en ce que :

- certaines opérations sur l'empreinte (telles que l'affectation et le contrôle d'empreinte) sont insérées explicitement dans le code du programme ;
- certaines opérations sur l'empreinte (telles que l'ajustement d'empreinte) sont effectuées explicitement par la machine d'exécution en fonction d'informations complémentaires au programme ;
- certaines opérations sur l'empreinte (telles que la mise à jour d'empreinte) sont effectuées implicitement par la machine d'exécution.

24. Procédé selon la revendication 12, caractérisé en ce que :

- dans le cas où des opérations de prise et/ou de contrôle d'empreinte sont effectuées par des appels de routine, le programme est accompagné d'une librairie qui implémente ces routines, ladite librairie étant pouvant être substituée par une implémentation particulière lors du chargement sur une plate-forme d'exécution ;
- dans le cas où les opérations de prise et de contrôle d'empreinte sont exprimées par des informations complémentaires au programme et où la plate-forme d'exécution ne sait pas et/ou ne peut pas et/ou ne veut pas exploiter ces informations, lesdites informations sont ignorées pour permettre l'exécution sans le contrôle d'intégrité.

25. Procédé selon l'une des revendications 12 et 20, caractérisé en ce que la machine d'exécution du programme dispose d'instructions spécialisées pour le calcul d'empreinte et/ou la mise à jour d'empreinte et/ou l'ajustement d'empreinte et/ou l'affectation d'empreinte et/ou le contrôle d'empreinte et/ou la sauvegarde d'empreinte à l'appel d'une routine et la restauration d'empreinte au retour d'une routine, ces instructions apparaissant explicitement dans le code du programme et/ou étant utilisées pour mettre en œuvre la machine d'exécution.

26. Système d'exécution permettant le contrôle d'intégrité d'exécution, caractérisé en ce que ledit système inclut un microprocesseur qui dispose d'instructions spécialisées pour le calcul d'empreinte et/ou la mise à jour d'empreinte et/ou l'ajustement d'empreinte et/ou l'affectation d'empreinte et/ou le contrôle d'empreinte et/ou la sauvegarde d'empreinte à l'appel d'une routine et la restauration d'empreinte au retour d'une routine, conformément au procédé selon l'une des revendications 1 à 25.